

LA-UR-21-28003

Approved for public release; distribution is unlimited.

Title: Using Computational Storage Devices: OpenMP/MPI and Charliecloud

Author(s): Janz, Jacob Benjamin
Goldstein, Justin James
Cunningham, Clyburn
Hammock, Charles Warren
Liu, Ralph
Rimerman, Mitchell

Intended for: Report

Issued: 2021-08-10

Disclaimer:

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by Triad National Security, LLC for the National Nuclear Security Administration of U.S. Department of Energy under contract 89233218CNA000001. By approving this article, the publisher recognizes that the U.S. Government retains nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

Using Computational Storage Devices: OpenMP/MPI and Charliecloud

Clyburn Cunningham IV, Justin Goldstein, Warren Hammock (USG),
Jacob Janz, Ralph Liu, Mitch Rimerman

Mentors: Shane Goff, Steve Poole, Kevin Bryant (USG)

Introducing Computational Storage Devices (CSDs)

- Computational Storage → Near-data processing
- Runs software where data resides
- Potential performance improvement
 - Offload tasks from host

Introduction

- Originally used Spark and HadoopFS
- Collected interesting results, but this method had its issues
 - Limited Application
 - Too much overhead to gauge CSDs' raw performance
- Solution? Rewrite our benchmarks without Spark:
 - Serial Python
 - Serial & Parallel C++ (Combinations of OpenMP & OpenMPI)

Why Serial Python?

Able to test on single core with no overhead.

Compare efficiency of different solutions.

- Implementations:
 - SparkDF & SparkSQL → Pandas (Dataframes) & Numpy (Matrices)
 - Natively written functions (no libraries)
 - Dataframes → Lists

Experiment Results: Running on One CSD

Function	100 MB	200 MB	500 MB	1 GB	5 GB
Count Lines	5.4598 e -5	5.3644 e -5	5.4836 e -5	5.4836 e -5	N/A
Sum of Column	0.1135	0.2281	0.5687	1.2115	N/A
Mean of Column	3.5763 e -5	3.5048 e -5	3.5048 e -5	4.0054 e -5	N/A
Grammian Matrix: AT*A	17.9477	35.6603	89.483	190.936	N/A
Normalize Column	5.3809	10.6566	25.6559	55.5248	N/A
Compute Mean	0.1138	0.2273	0.5676	1.2162	N/A
Compute Std Dev	3.6919	7.173	17.9616	38.0247	N/A
Count Digits	6.668	6.4407	16.0995	34.4509	N/A
Measure Shannon Entropy	343.624	650.1484	1699.7029	3576.3302	N/A
Total Elapsed Time	6.8031 Minutes	13.1948 Minutes	34.1343 Minutes	71.9462 Minutes	N/A

Where to Go From Python?

Python's Shortcomings

- Running in “parallel” is less than ideal in native Python

Using Python's Multithreading Libraries?

- Typically accelerates one machine
- C++ implementation would be more thorough

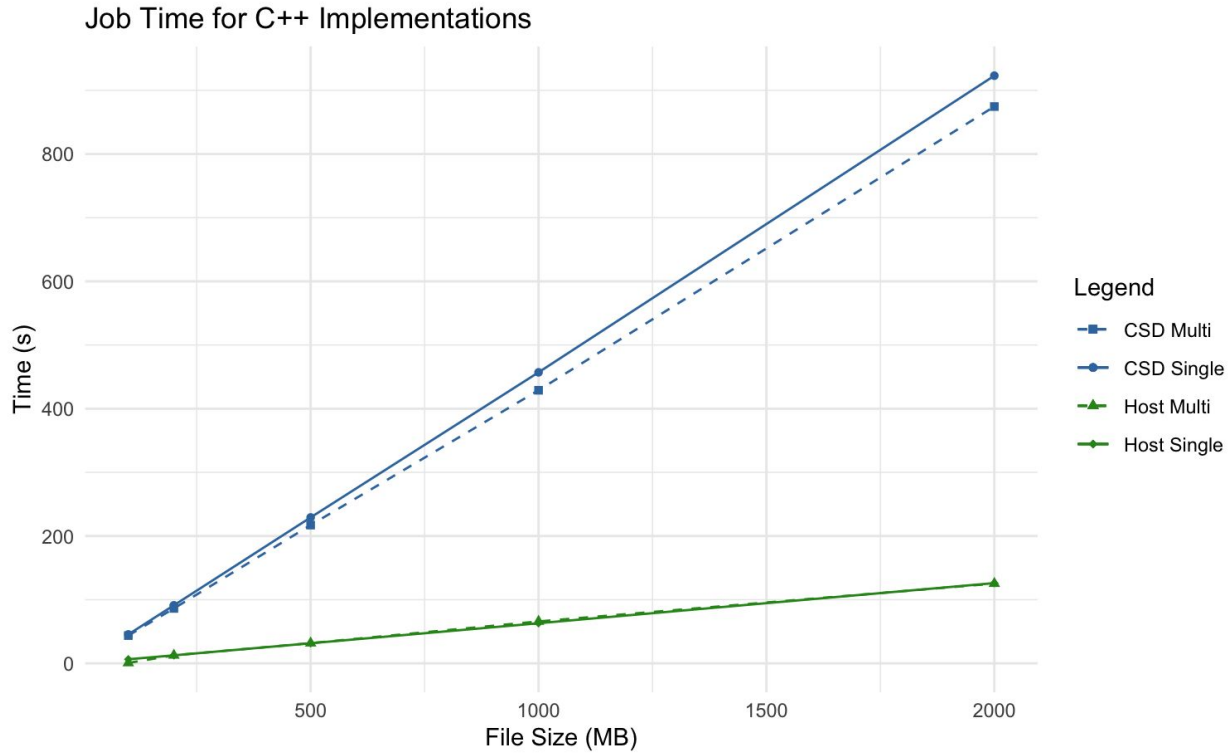
Duplicating Spark Tests in C++

- C++ is “lower level” than Pyspark or basic Python
 - Lets us get a better understanding of CSDs baseline performance
- Basic C++ Implementation is a reimplemented version of our Spark program, with a single-threaded and a multi-threaded version using OpenMP

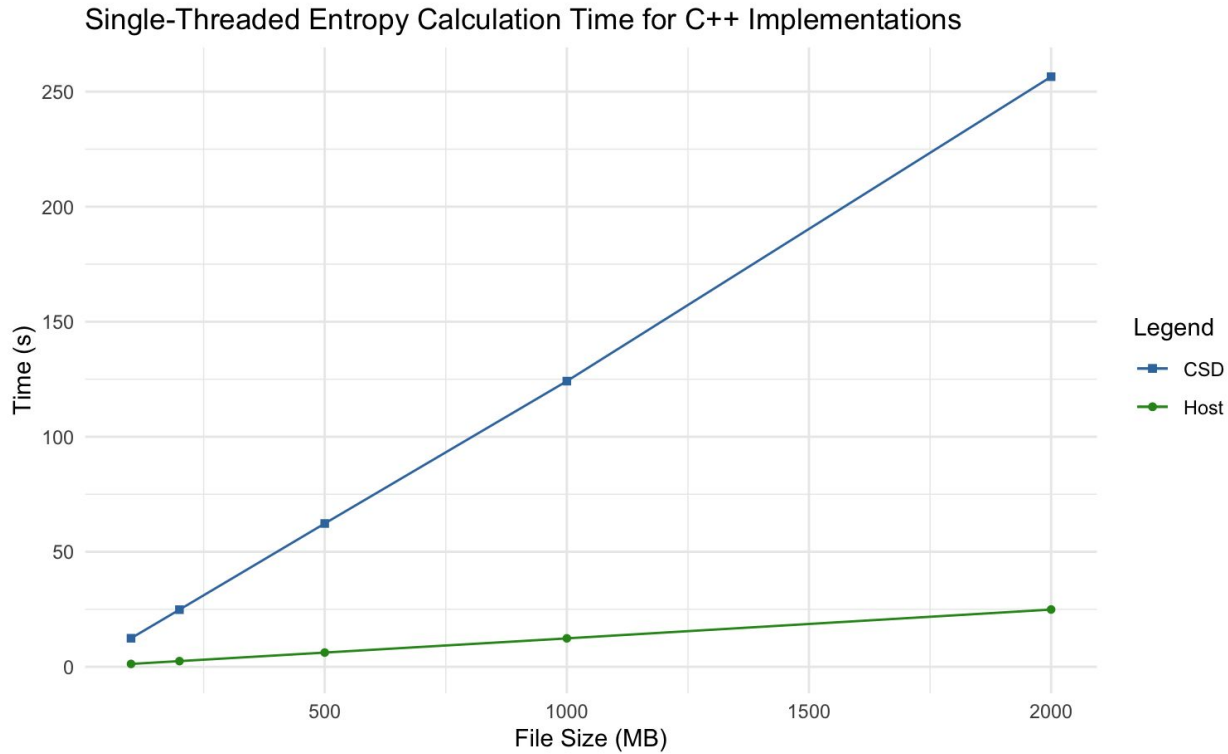
```
// get sum of normalizedVector
auto sumNormVectTimeStart = std::chrono::high_resolution_clock::now();
int normalizedVectorSum = 0;
for (int i = 0; i < normalizedVector.size(); i++)
{
    normalizedVectorSum += normalizedVector[i];
}
auto sumNormVectTimeEnd = std::chrono::high_resolution_clock::now();
std::cout << "Sum of third row normalized is: " << normalizedVectorSum << std::endl;
```

```
// get sum of normalizedVector
auto sumNormVectTimeStart = std::chrono::high_resolution_clock::now();
int normalizedVectorSum = 0;
#pragma omp parallel for
for (int i = 0; i < normalizedVector.size(); i++)
{
    #pragma omp atomic update
    normalizedVectorSum += normalizedVector[i];
}
auto sumNormVectTimeEnd = std::chrono::high_resolution_clock::now();
std::cout << "Sum of third row normalized is: " << normalizedVectorSum << std::endl;
```

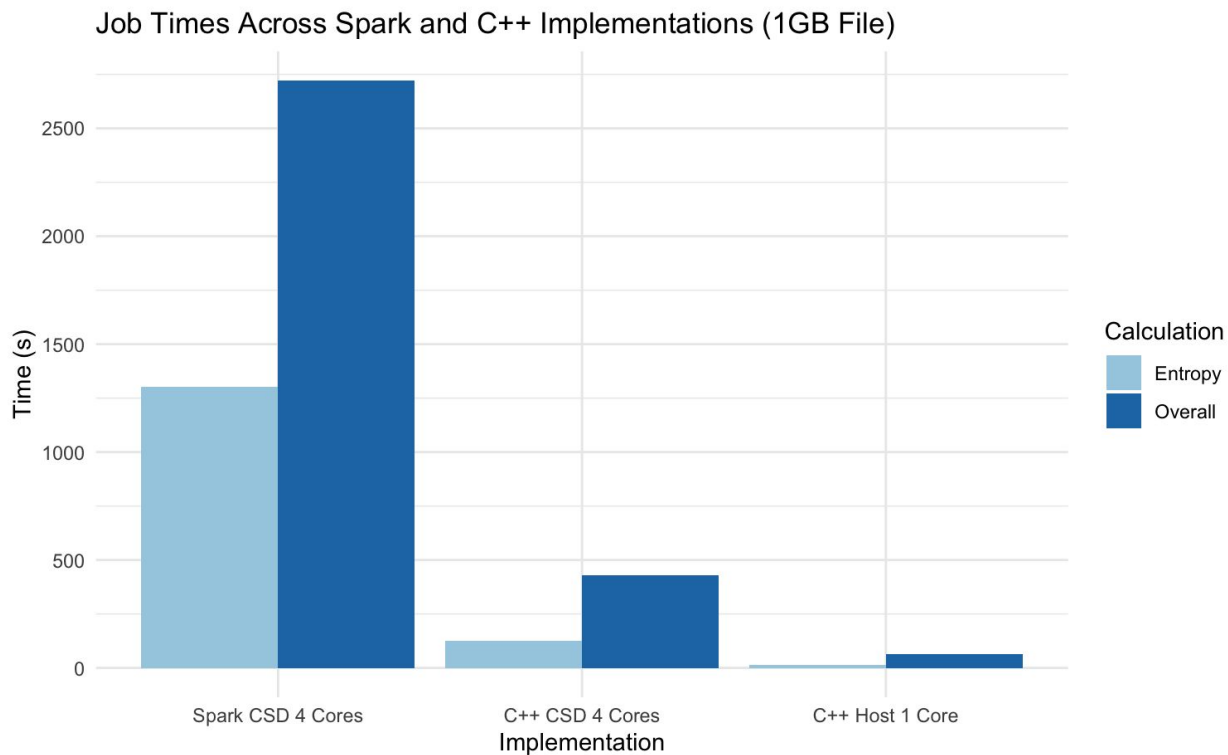
Results



Results contd.



Results contd.



C++ Conclusions and Thoughts

- Compared to Spark and Python, C++ implementation is *a lot* faster
 - Caveat: an expert with Spark or Python would likely be able to improve the performance of those implementations
- Computational power of our CSDs seem to be much lower than the host machine
 - Using all 4 cores of a single CSD, the job takes ~6.8x longer than using just one core on the host machine.
 - Host also seems to scale better with increasing file size
- Resulting Question: When, if ever, would it make sense to use CSDs for compute rather than a much-faster host?

Host (1.5GHz) and CSDs (1GHz)

Host: 128GB RAM (8GB swap)

Architecture: x86_64

CPU(s): 64

Thread(s) per core: 2

Core(s) per socket: 32

Socket(s): 1

CSD (x8): 5.8 GB RAM

Architecture: aarch64

CPU(s): 4

Thread(s) per core: 1

Core(s) per socket: 4

Socket(s): 1

How to Offload Selected Operations?

Disclaimer: Our test was done using host system and 1 csd node (not the full 8 supported). This analysis applies specifically to the operations used in this experiment.

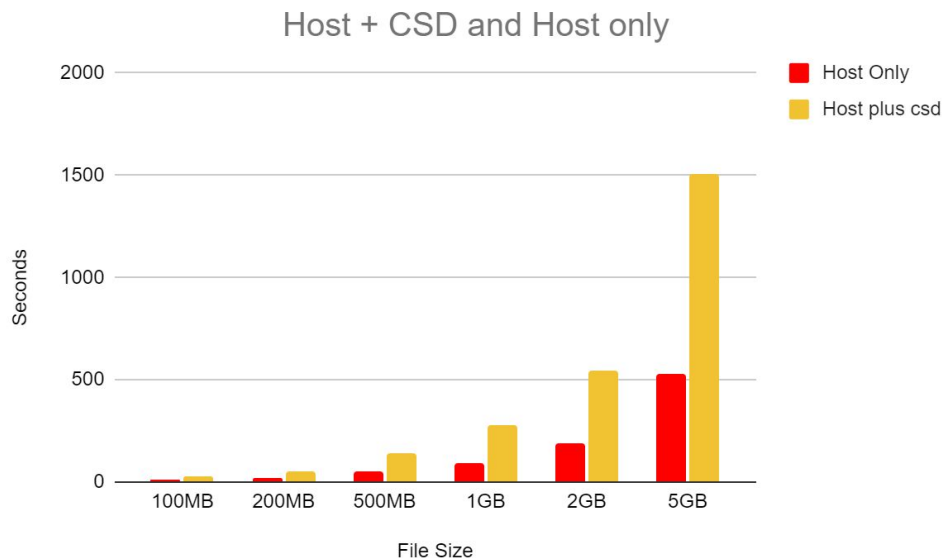
Why use MPI?

Tests: Quickest scalable operations:

- Compute mean (constant time)
- Normalized Compute sum
- Normalized Compute standard deviation
- Normalized Count frequency of digits

When does it make sense to distribute our operations to the CSD? Host and CSD reading in log file

- Tool used: stress-ng --cpu 64 --vm 1 --vm-bytes 95% (stressed RAM and core count)
- Stressed Host tested with mounted CSD storage.
- No Stress CSD tested with mounted CSD storage.



Can message passing be used to decrease csd vector build time

Issue:

- Most expensive operations for the CSD was to read file and build vector.
- Host completes those operations in 5.86(s)(stressed) 3.91(s)(no stress)
- CSD completes those operations in 23.75(s)

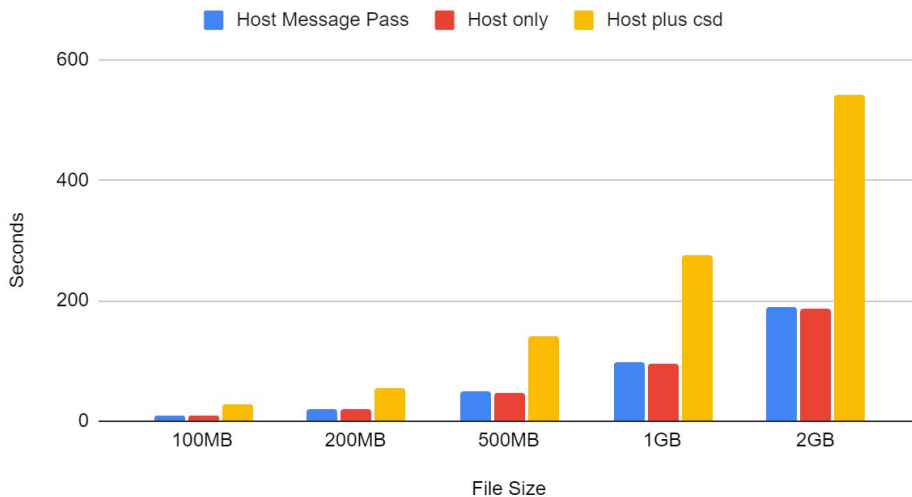
Test:

- 100MB/200MB/500MB/1GB/2GB log file.
- The host reads file from CSD storage and creates vector. Host will then message pass vector to csd.
- See if there is an decrease in overall time for csd to complete its operations.
- Additional parameter for mpirun --mca btl_tcp_if_include flannel.1 (includes interface)

Offloading operations passing vector to CSD

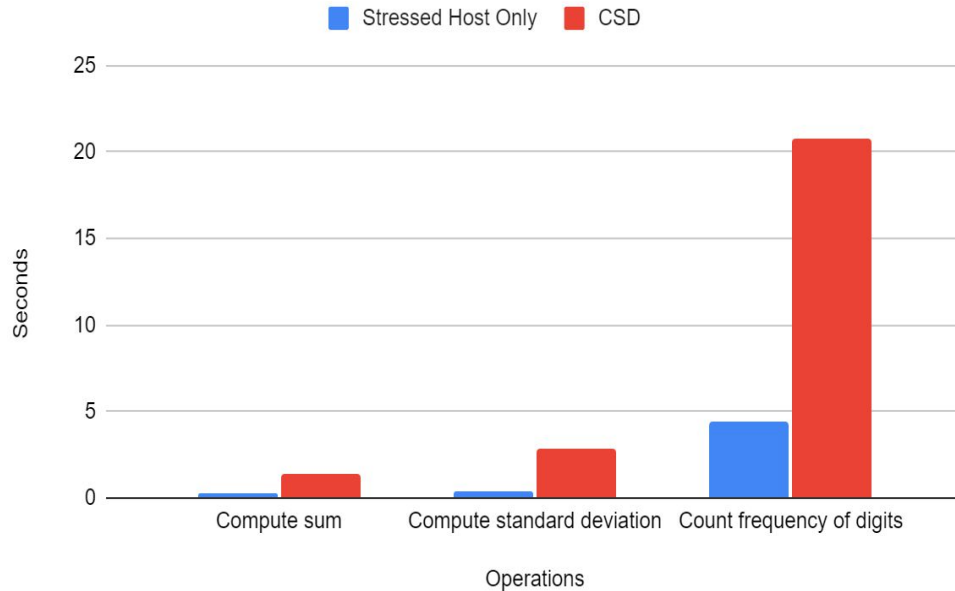
- Tool used: stress-ng --cpu 64 --vm 1 --vm-bytes 95% (stressed RAM and core count)
- Stressed Host tested with mounted CSD storage.
- No Stress CSD tested with mounted CSD storage.

Host + CSD(Message), Host only, and Host + CSD



Still does not make sense on a per operation comparison

Stressed Host Only and CSD Operation Times



- Operation costs on a 1GB data log.
- Even after vector is in memory, the csd still executes the operation significantly slower than the stressed host test.
- Future work needs to be done with a focus on small operations. CSDs seem to be of more use in smaller operations on smaller files.

Future work for passing information

- Further investigate MPI's usage for communication.
- Need to develop a better way for host and csds to share storage.
- Create a pooled storage for CSDs, possibly ZFS.
- Data filtering (encrypt/decrypt)

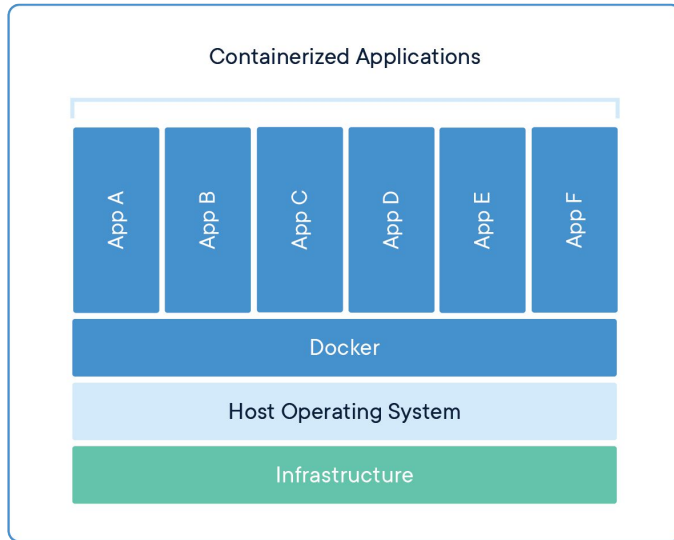


CSDs with Charliecloud

About Charliecloud
Background on experiments
Analysis of results



About Charliecloud



- Bring your own software stack
 - Containers
 - Container images
 - Code
 - System tools
 - Runtime
 - Settings
- Charliecloud Images
 - Few permissions
 - Minimally affect cluster resources

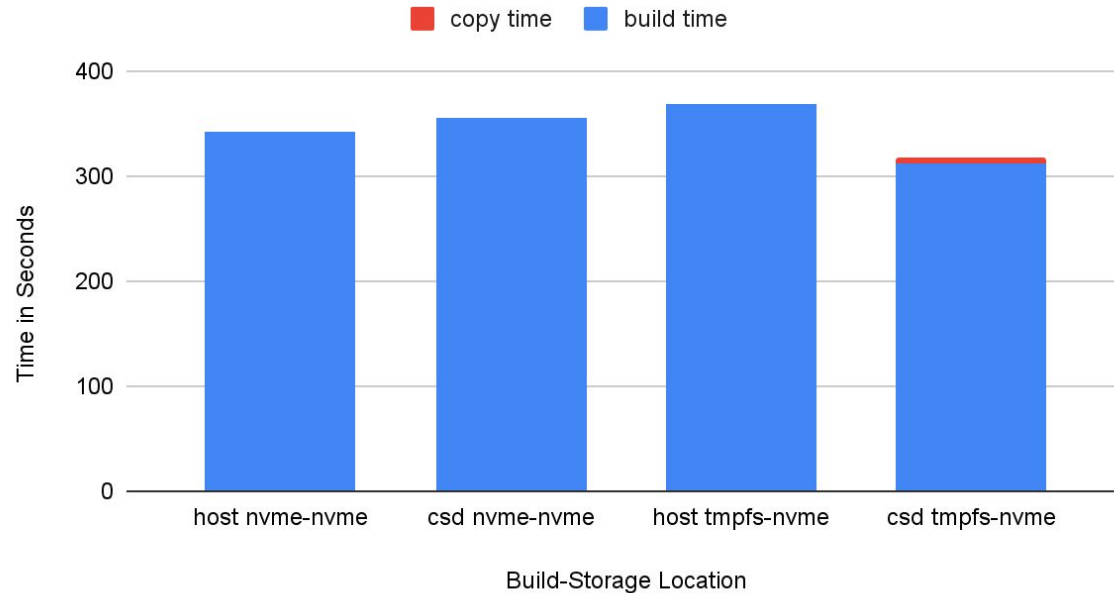
Experiments

Build Location	Storage Location
Host NVME	Host NVME
Host tmpfs	Host tmpfs
CSD NVME	CSD NVME
CSD tmpfs	CSD tmpfs

- Typical workflow: Build image on a compute node
 - (Inefficient!)
- Research Question: What is the best filesystem to store user images on in a cluster environment?
 - Compare small CSD to big host
 - Compare big host to LANL's fog (later)

CSDs out-perform host on small image?

Time of Charliecloud Build and Storage Host vs CSD



Experiments

Build Location	Storage Location
NVME	NVME
tmpfs	NVME
NFS	NFS
LUSTRE	LUSTRE
tmpfs	LUSTRE
tmpfs	NFS

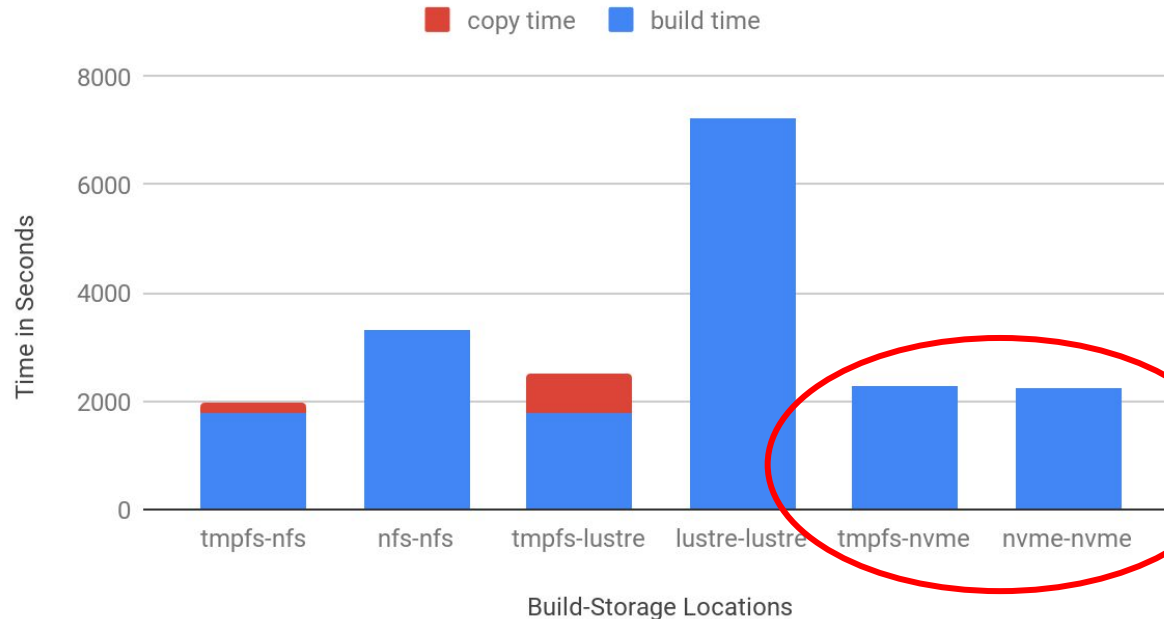
- How does our host with NVME compare to a LANL production setup?

- Lustre on Fog vs
- NFS on Fog vs
- NVME on our host

Our host

NVMe vs Other Filesystems

Time of Charliecloud Image Build and Storage



Conclusions and Next Steps

- Future work on variability across runs
 - Implications for scaling to larger container image builds
- Viability of CSDs for medium term storage
- Potential use case for CSDs with Charliecloud
 - Envisioning a new user workflow

Overall times to complete all operations per data size

Method	Spark		Python	C++			
	1 CSDs	8 CSDs	Serial on CSD	Serial on CSD	Multithread on CSD	Host stressed and CSD	Host stressed
100MB	N/A	N/A	408 s	45.47s	43.94s	36.83s	9.82s
200MB	N/A	N/A	792 s	90.61s	87.40s	72.51s	19.36s
500MB	N/A	N/A	2,048 s	229.10s	217.09s	181.97s	48.16s
1GB	2759.17s	542.44s	4,317 s	457.74s	432.54s	358.16s	94.61s
2GB	N/A	N/A	N/A	929.60s	870.97s	714.72s	187.58s